Transforming Object-oriented Models to Process-oriented Models

Guy Redding[1], Marlon Dumas[1], Arthur H.M. ter Hofstede[1] and Adrian Iordachescu[2]

[1] Queensland University of Technology, Brisbane, Australia
{g.redding, m.dumas, a.terhofstede}@qut.edu.au
[2] Shared Web Services Pty Ltd, Sydney, Australia
adrian@sws.com.au

Abstract. Object-oriented modelling is an established approach to document the information systems. In an object model, a system is captured in terms of object types and associations, state machines, collaboration diagrams, etc. Process modeling on the other hand, provides a different approach whereby behaviour is captured in terms of activities, flow dependencies, resources, etc. These two approaches have their relative advantages. In object models, behaviour is split across object types, whereas in process models, behaviour is captured along chains of logically related tasks. Also, object models and process models lend themselves to different styles of implementation. There is an opportunity to leverage the relative advantages of object models and process models by creating integrated meta-models and transformations so that modellers can switch between these views. In this paper we define a transformation from a meta-model for object behavior modeling to a meta-model for process modeling. The transformation relies on the identification of causal relations in the object model. These relations are encoded in a heuristics net from which a process model is derived.

Key words: Process model, object model, model transformation

1 Introduction

Object modelling and process modelling are two established approaches to describe information systems [1]. Each of these approaches adopts a different perspective and has its own way of thinking. Modelling an information system in terms of objects leads to the definition of object types, associations, intra-object behaviour and inter-object interactions, which are captured using notations such as UML class, state and collaboration diagrams [2]. Object models group related data and behaviour into classes, thus promoting modularisation and encapsulation. Purported advantages of this approach include reuse and maintainability.

Meanwhile, process models are structured in terms of activities (which may be decomposed into sub-processes), events, control and data-flow dependencies, and associations between activities and resources. BPMN [3], UML activity diagrams, BPEL [4] and YAWL [5] are examples of notations that capture the

behaviour of a system in a process-oriented manner at various levels of details. Process models provide a holistic view on the activities and resources required to achieve a goal. Accordingly, they lend themselves to analysis through simulation and other quantitative analysis techniques, and they have proven instrumental in enabling communication between business and IT stakeholders [6].

Moreover, object modelling and process modelling typically lead to different implementation styles. Whereas object modelling lends itself to implementation in an object-oriented programming environment, process models naturally lead to workflow applications or other types of process-aware information systems.

There is an opportunity to reconcile object-oriented and process-oriented approaches to information systems engineering in order to benefit from their relative strengths. Each of these modelling approaches adopts a different perspective. Information captured in one approach may be missing in the other approach. For example, a class in an object model may contain references to activities (e.g. in a sequence diagram), but objects are predominately state-centric and do not explicitly define activities or the control flow relations between them. Likewise, a process model contains implicit references to states (cf. the event-driven and the deferred choice in BPMN and YAWL respectively) or to classes representing resources, but a process model is predominately activity-centric.

Figure 1 shows the typical phases and deliverables involved in the object-oriented development approach (OODA) and in the process-oriented development approach (PODA). In this paper, we investigate how to bridge the deliverables produced by the design phases of these two approaches. Specifically, we present a transformation from detailed object behaviour models to process models. The transformation relies on the identification of causal relations in the object model. These relations are encoded in a causal matrix (also called heuristics net) from which we derive a process model represented in YAWL.

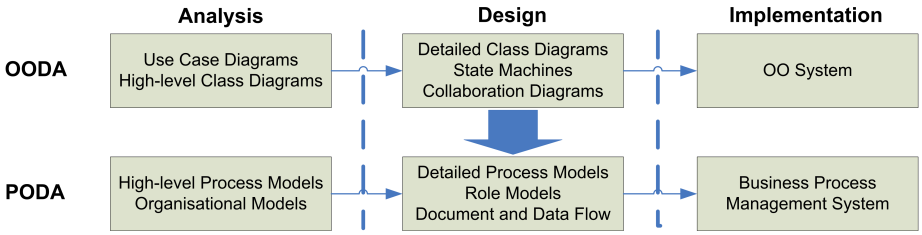

Fig. 1. Transforming an Object-oriented to a Process-oriented Approach

The paper is organised as follows. Section 2 introduces a motivating example. Section 3 defines a meta-model for object behaviour modelling. Section 4 introduces an algorithm to transform an object behaviour model into a process model. Section 5 discusses related work and Section 6 concludes the paper.

2 Example

In this section we introduce an example of a process that we have used as a test scenario. The example deals with a process for inspection and maintenance of heavy equipment such as open mine excavators and shipping container cranes. Such equipment is subjected to inspections at regular intervals when a number of issues requiring maintenance may be raised with the equipment. Depending on the severity of an issue and the criticality of the equipment some issues will be determined to be resolved with more urgency than others. The application domain is presented as a high-level class diagram in Figure 2. Each class may have one or more state machines as discussed later.

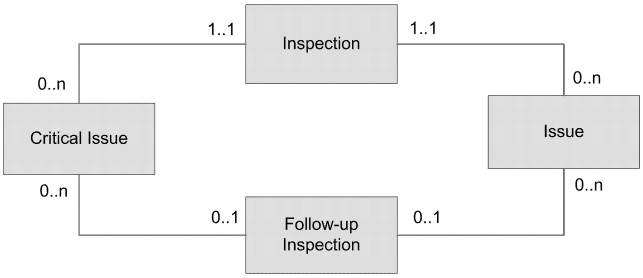

Fig. 2. Asset Maintenance Process – High-level class diagram

An *inspection* for a particular piece of equipment may uncover zero or more *issues* to be resolved. These are added to the set of existing *issues* that can be derived for that piece of equipment so that the main *inspection* can only be completed after all *issues* have been resolved and completed. A *critical issue* may also be detected during an *inspection* or *follow-up inspection*. These *critical issues* are identified separately due to the elevated need to have them resolved. An *issue* may raise a number of *follow-up inspections*, which in turn may lead to new (critical) *issues* being raised, and so on. The multiplicity between classes such as 1:1 and 0:n specifies the number of instances that a class interacts with at runtime. It should be noted that the reason that an *inspection* is modelled as a separate entity to a *follow-up inspection* is that in this scenario these classes have different state machine lifecycles, as do an *issue* and *critical issue*.

3 Object Behaviour Meta-Model

In order to present our transformation approach, we first need to agree on meta-models for representing object behaviour models and process models. To represent process models, we use the YAWL language as discussed in the next section. Meanwhile, to represent object models we adopt a meta-model inspired

by FlowConnect [7], a system that supports the development of software applications based directly on executable object behaviour models. The meta-model is presented as an Object Role Model (ORM) [8] in Figure 3.

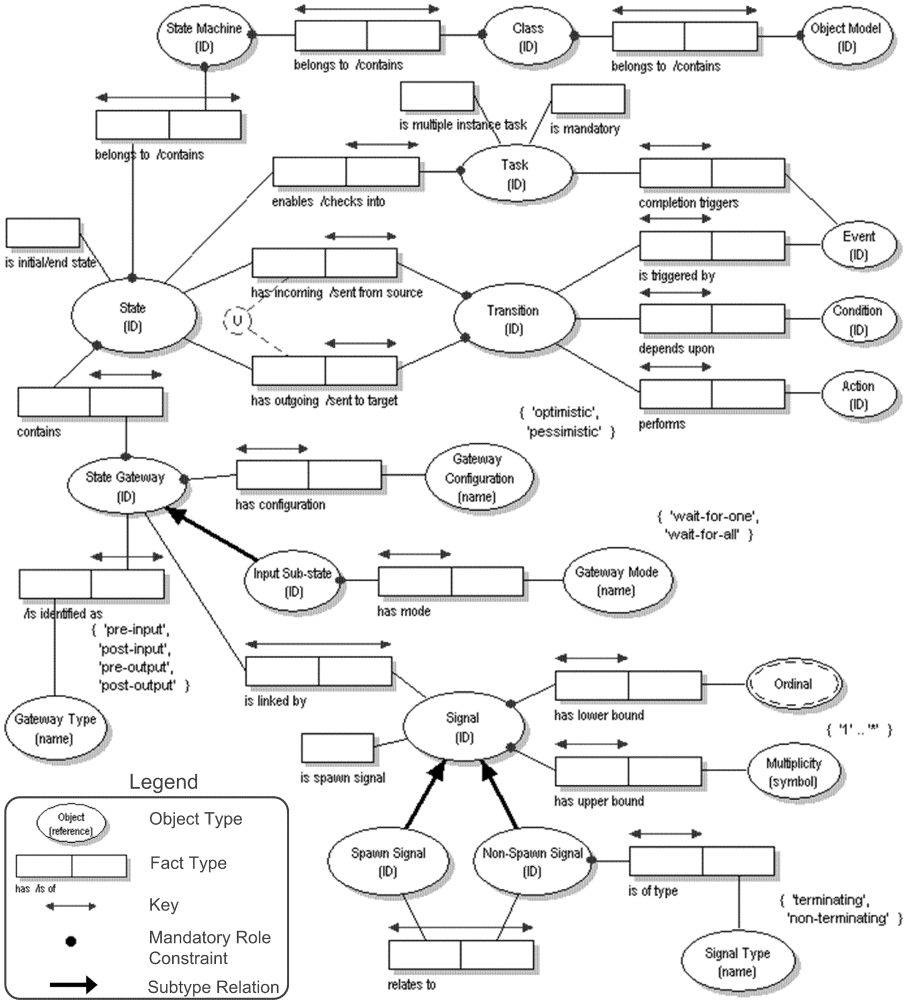

Fig. 3. Object Behaviour Meta-Model

FlowConnect is an attractive source meta-model for our proposal for two reasons. Firstly, FlowConnect seamlessly integrates concepts from UML state diagrams with concepts from UML sequence diagrams, allowing us to capture both intra-object and inter-object behavior in the same model. Secondly, the FlowConnect-based meta-model is a representative of other object-oriented meta-

models (e.g. Proclets [9], Merode [10], OCoN [11]), thus it is possible to adapt the results presented here to other meta-models.

At the highest level an **object model** is a container for all classes in an object-oriented model. We define a class as a "cluster" of related states that share some common context. An example of shared context are states that belong to an *inspection*. Grouping these states together allows control flow associations to be made between them and an *inspection* begins to take shape. The object model contains one or more **classes** that contain one or more **state machines**. A state machine contains one or more **states**. A state models a moment in a process lifecycle where the context of a process can be distinguished and named, e.g. *Distribute Report*, *Load Data* or *Test Structural Strength*.

A **transition** connects the output of a state (the source state) to the input of another state (the target state) in the same state machine. Transitions may have an optional Event-Condition-Action (ECA) rule. The occurrence of an **event** will cause the transition labeled by that event to be performed. A transition can have a **condition** associated with it that must be satisfied before the transition can occur. When a transition is performed it may also execute an **action**. The details of an ECA rule language are not specified since this is out of scope. Transitions are indicated on a link between two states by an arrow with an open arrowhead.

Each state contains three sub-states; a **pre-gateway**, **main processing sub-state** and **post-gateway**. The main processing sub-state is where work is completed in a state and it contains zero or more atomic **tasks**. The pre- and post-gateways are the entry and exit points of the main processing sub-state respectively. A pre-gateway is entered when the state it belongs to has been entered by an incoming transition. A gateway may send signals to other state gateways and receive (wait for) signals from other state gateways using an input sub-state to receive incoming signals and an output sub-state to send outgoing signals. The order in which these signals are sent or received depends upon the gateway configuration, i.e. a *pessimistic* gateway will wait to receive all signals it expects before sending any whereas an *optimistic* gateway sends signals before waiting to receive any. An input sub-state also has a mode to specify whether it should wait for the first signal (*wait-for-one*) or all signals (*wait-for-all*) before control flow will be released.

A **signal** establishes a one-way connection between state gateways that belong to two different state machines. In contrast to a transition, a signal does not have an ECA rule. There are three types of signal that are distinguished by the arrowhead on a link between two gateways: **spawn** (i.e. creates a new state machine) indicated by a double-filled arrowhead, **finish** (i.e. terminates a state machine) indicated by a double-empty arrowhead and **message** (i.e. non-terminating) indicated by a single solid arrowhead. A signal has a lower and upper bound, which are the minimum and maximum number of times it can be sent, i.e. a spawn signal with a lower bound of 1 and upper bound of 5 can create between 1 and 5 objects.

Some signals occur in response to, or following another signal. For example, a non-terminating (message) signal *Sig02* can only occur following a spawn sig-

nal *Sig01*. Accordingly, the meta-model includes an association between signals which form a **relationship**.

As an example, a fragment of a state machine corresponding to the Inspection class, as well as two related state machines corresponding to the Critical Issue and the Issue classes are shown in Figure 4.

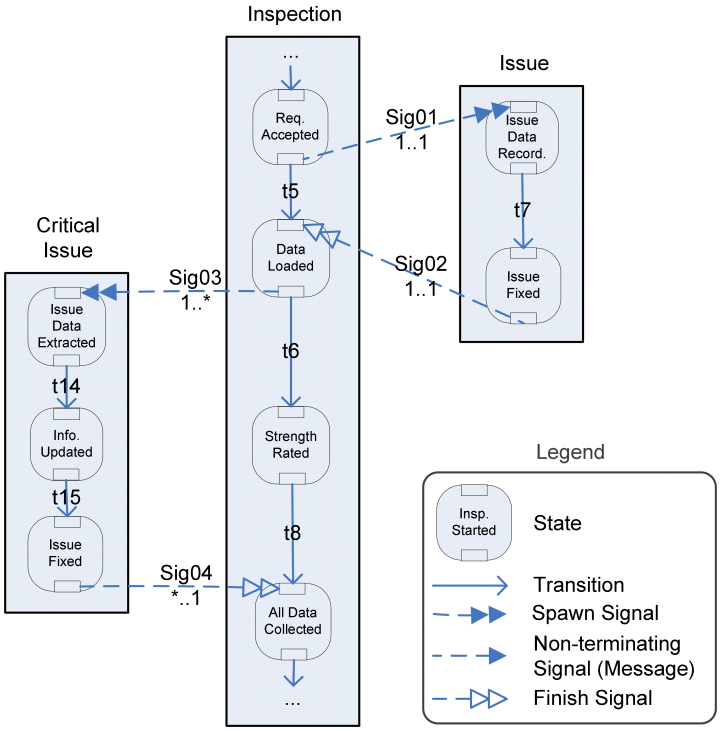

Fig. 4. Example of an Object Behaviour Model

4 From Object Behaviour Models to Process Models

In this section we introduce a proposal to map an object behaviour model to a process model. We use the asset maintenance process as an illustration of this proposal. The essence of the proposal is to analyse the object behaviour model in order to extract a set of elementary causal dependencies between events and signals. These elementary causal dependencies are represented as a *causal matrix*, also known as a *heuristics net* [12]. The idea of using a heuristics net comes from the ProM framework [13], where heuristics nets are used as an intermediate representation to construct a Petri net from an event log.

A heuristics net is composed of a set of transitions, which we call "tasks" to put them in the context of this paper. Each task has an *input* and an *output*. The input of a task T represents the different ways in which task T can be started. Concretely, the input of a task is a set. If this set is empty, it means that the task can be started even if no other task has been completed (i.e. this is the initial task in the process model). If the input of a task is not empty, it contains one of several *disjunctions*. Each of these disjunctions should be read as an "Or" of several tasks. For example, a disjunct { F,B,E } means that either task F or task B or task E have completed. The different disjunctions in an input are implicitly linked through an "And", meaning that each disjunct must be satisfied before the target task can be started. For example, the input of task D is { {F,B,E}, {E,C}, {G} }. For task D to be executed, either F or B or E must be completed, and either E or C must be completed, and G must be completed.

Symmetrically, the output of a task determines which other tasks can be executed after a given task completes. An empty output denotes a final task in the process. Meanwhile, a non-empty output must be read as a set of disjuncts. For example, a disjunction of the form { A,B,C } means that either A or B or C can be executed. An output can contain multiple disjunctions. The output of task A is { {F,B,E}, {E,C}, {G} }, which means that after A completes, either F or B or E will be executed, and either E or C will be executed, and G will be executed. Readers familiar with Petri nets will recognise that a heuristics net is a Petri net of a particular form. The transformation procedure depicted in Figure 5 consists of these steps:

I - Generate a heuristics net from an object model/state machine diagrams.
II - Generate a Petri net from a heuristics net.
III - Transform the Petri net into a YAWL process model.

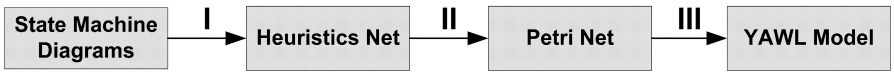

Fig. 5. An Overview of the Transformation Procedure

Below, we present an algorithm that automates **Step I**. For each state in an object model, this algorithm generates two tasks corresponding to the pre- and post-gateway. In other words, each pre- or post-gateway in the object model will lead to one task in the generated heuristics net. Because the main-processing sub-state has no more than one input (from the pre-gateway) and one output (to the post-gateway) it is not represented explicitly in the heuristics net.

Algorithm 1 takes as input an object model and produces the corresponding heuristics net. The algorithm iterates over each state gateway in order to generate an input set (preTask) and output set (postTask). Because there is a one-to-one mapping between state gateways and tasks, the algorithm treats

them interchangeably, meaning that it uses the identifiers of gateways in the source object model as identifiers of tasks in the generated heuristics net.

The following auxiliary functions are used in Algorithm 1:

- *states* : ObjectModel → Set of State, is the set of states in an object model.
- *pre, post* : State → Gateway, yields the pre or post gateway of a state.
- *inputTransitions, outputTransitions* : State → Set of Transition, yields the set of input/output transitions.
- *source, target* : Transition → State, yields a transition's source/target.
- *inputSignals, outputSignals* : Gateway → Set of Signal, yields a gateway's input/output signals.
- *mode* : Gateway → GatewayMode, yields a gateway's mode.
- *explode* : Set of Signal → Set of Set of Signal. explode($\{e_1, e_2, \ldots, e_n\}$) = $\{\{e_1\}, \{e_2\}, \ldots, \{e_n\}\}$.

Algorithm 1: Generation of a Heuristics Net

Input: om : ObjectModel
Output: preTask, postTask : Task → Set of Set of Task
predecessors, successors : Set of Gateway
foreach *s* ∈ *states(om)* **do**
  predecessors := { post(source(t)) | t ∈ inputTransitions(s) };
  successors := { pre(target(t)) | t ∈ outputTransitions(s) };
  preInputSignals := { source(g) | g ∈ inputSignals(pre(s)) };
  preOutputSignals := { source(g) | g ∈ inputSignals(post(s)) };
  postInputSignals := { target(g) | g ∈ outputSignals(pre(s)) };
  postOutputSignals := { target(g) | g ∈ outputSignals(post(s)) };
  if *mode(pre(s)) = wait-for-one* **then**
    preTask(pre(s)) := { predecessors, preInputSignals };
  else
    preTask(pre(s)) := { predecessors } ∪ explode(preInputSignals);
  postTask(pre(s)) = { { post(s) }, postInputSignals(s) };
  if *mode(post(s)) = wait-for-one* **then**
    preTask(post(s)) := { { pre(s) }, preOutputSignals(s) };
  else
    preTask(post(s)) := { { pre(s) } } ∪ explode(preOutputSignals(s));
  postTask(post(s)) := { successors } ∪ explode(postOutputSignals(s));
end

To analyse the inbound and outbound causal dependencies of a gateway, we conceptually decompose each gateway into two parts: the input and the output. The input corresponds to the signals the gateway has to wait for, while the output corresponds to the signals it has to send out. Figure 6 depicts the decomposition of the pre- and post-gateways of a state into an input and output part.

The input and output sets are generated as follows. The pre-gateway input set is the union of the source of each incoming transition with the source of each incoming signal, depending on the gateway mode (i.e. *wait-for-one* or *wait-for-all*). If the gateway mode is *wait-for-one* then the preTask set is the set of

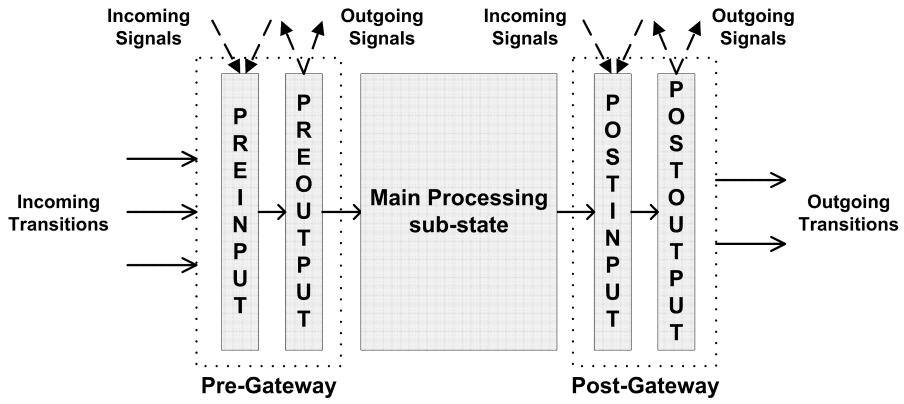

Fig. 6. Pre- and Post-Gateways of a State

input transition sources (predecessors) and the set of pre-gateway input signal sources. However if the gateway mode is *wait-for-all* then the preTask set is the union of the set of input signal sources converted to a set of set of signals by the *explode* function with the set of predecessors. The postTask set consists of the post-gateway and the targets of all outgoing signals sent from the pre-gateway. The procedure for constructing the input and output sets for a post-gateway is symmetric to the corresponding procedure for a pre-gateway. After completion of the algorithm a heuristics net is constructed by inserting the input and output set as individual rows in the net.

A heuristics net is a 'flat' representation of a process since it does not capture sub-processes. When converting a heuristics net to a YAWL net, it is desirable to incorporate sub-processes. This can be achieved by identifying "sub-process delimiters" in the object behaviour model. These delimiters are the points where an instance of a state machine is created, and the point(s) where a state machine returns a terminating signal to its parent. In the absence of message signals between the parent and child state machines, the region between a spawn signal and the finish signal in the resulting YAWL net will be a single-entry-single-exit (SESE) region.

A SESE region in the resulting YAWL net whose entry point corresponds to a spawn signal with a multiplicity of one, and whose exit point correspond to a matching finish signal is converted into a YAWL composite task (i.e. the YAWL construct for capturing sub-processes). Similarly, a SESE region whose entry point corresponds to a spawn signal with a multiplicity greater than one is converted into a YAWL multiple instance composite task (i.e. a sub-process that is executed multiple times concurrently). This procedure of identifying sub-process delimiters in an object model and restoring the delimiters back in the resulting YAWL net allows us to obtain more modular YAWL nets.

In **Step II** the heuristics net is passed to the Heuristics Net conversion tool in ProM to obtain a Petri net. **Step III** is performed using a workflow net

conversion plugin in the ProM framework to combine the sub-process delimiters with the derived Petri net to create an 'unflattened' YAWL process model where properties such as task multiplicity, state machine multiplicity and sub-process definitions are restored in the YAWL model as shown in Figure 7. This step is merely a syntactic transformation that aims to exploit the constructs in YAWL because any Petri net can be seen as a YAWL process model. The complete model is a process-oriented view of the control flow between the input and output gateways for every state in an object model.

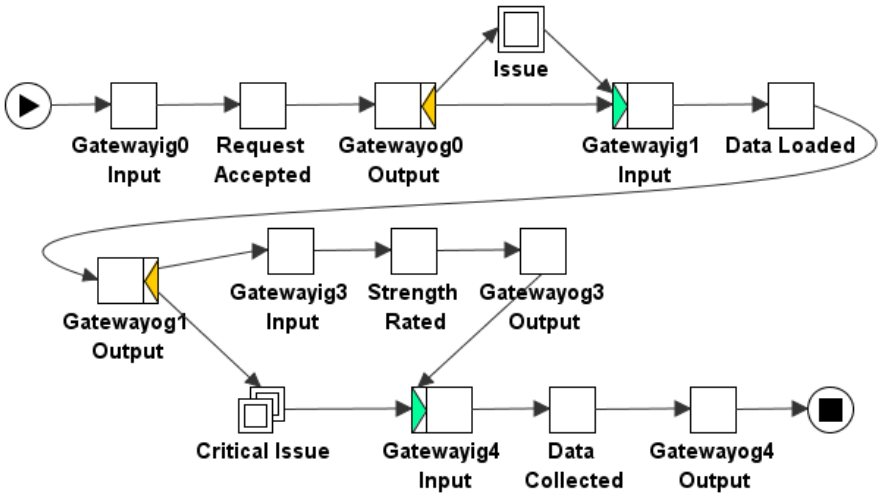

Fig. 7. Resulting YAWL Process Model

Implementation

The proposal has been implemented in Java on top of the Eclipse platform.[3] The tool includes a graphical editor for object behaviour models and a module that support the transformation of object behaviour models to YAWL nets. The tool implementation relies on libraries from the ProM framework to perform the transformation from heuristic nets to Petri nets and from Petri nets to flat YAWL models. Subsequently, these YAWL models are unflattened as explained above.

The modelling tool and the model transformation technique have been tested using the asset maintenance example. Future plans for this tool includes adding support for data flow modelling and resource modelling and enhancing the model transformation technique to cater for these modelling perspectives.

[3] http://www.eclipse.org/platform/

5 Related Work

Object-oriented (OO) design methodologies that use the UML to design and develop Information Systems have been proposed such as OCoN [11]. These proposals link UML diagrams to phases of the process development lifecycle to produce a schema as output at the conclusion of the lifecycle. Our proposed approach would extend these design methodologies and allow process analysts and designers to produce a completely process-oriented view of an OO model. FlowConnect [7] and Proclets [9] are examples of an OO system and notation.

Reijers et al. proposed a methodology for Product-Based Workflow Design (PBWD) that presented an analytical clean-sheet approach for process design specified by the bill-of-material for products that are affected by the process [14]. Since PBWD focuses on the bill-of-material, we consider this is an OO modelling approach. In PBWD there is no notion of object life-cycles, which is an area covered by artefact-centric process modelling [15]. The artefact-centric approach unifies data and process in an "Operational Specification" but does not consider converting these specifications to other modelling representations.

An architecture for mapping between OO and activity-oriented process modelling approaches has been proposed by Snoeck et al. [10]. This architecture maps OO development to process modelling. Object associations and business rules are captured using object-relationship diagrams and an object-event table models the behaviour of domain objects, which are similar to our mapping artefacts.

6 Conclusions and Future Work

Object technology is a mainstream approach to implementing Information Systems. Mainstream object-oriented analysis and design practices (e.g. those based on UML) are based on concepts of objects whose structure is captured as classes and whose behavior and interactions are captured as state machines, sequence diagrams and similar notations. On the other hand, recent trends have seen an uptake of approaches to Information Systems engineering that treat processes as a central concept throughout the development lifecycle.

The co-existence of these two approaches may lead to situations where a project starts with a model corresponding to one approach and needs to switch to a model corresponding to the other approach. In this paper, we have proposed an approach to help bridge these differences in terms of the control flow logic and discussed how the conversion technique has been implemented.

Future work will continue on the topic of transforming object-oriented models to process-oriented models and vice-versa. There is a need to cover not only the control-flow aspects as outlined in this paper, but also data flow and resource allocation. We also note that a similar problem arises in the opposite direction, i.e. moving from a process-oriented to object-oriented design for the purpose of implementing process-oriented design models using object-oriented technology. Therefore another future challenge will be a proposal of a reverse transformation from process-oriented to object-oriented models.

Acknowledgement This work is supported by an Australian Research Council Linkage grant (LP0562363) co-funded by Shared Web Services Pty Ltd.

References

1. Kueng, P., Bichler, P., Kawalek, P., Schrefl, M.: How to compose an object-oriented business process model? In: Proceedings of IFIP TC8, WG8.1/8.2 working conference on method engineering, London, UK, Chapman & Hall, Ltd. (1996) 94–110
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley Professional (1998)
3. Object Management Group: Business Process Modelling Notation, Ver 1.0. http://www.bpmn.org (2006)
4. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.1. http://dev2dev.bea.com/webservices/BPEL4WS.html (2003)
5. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Information Systems **30** (2005) 245–275
6. Becker, J., Kugeler, M., Rosemann, M.: Process Management. Springer (2003)
7. Shared Web Services Pty. Ltd.: FlowConnect Model. (August, 2003)
8. Halpin, T.: Information modeling and relational databases: from conceptual analysis to logical design. Morgan Kaufmann Publishers Inc. (2001)
9. van der Aalst, W.M.P., Barthelmess, P., Ellis, C.A., Wainer, J.: Proclets: A Framework for Lightweight Interacting Workflow Processes. International Journal of Cooperative Information Systems **10** (2001) 443–481
10. Snoeck, M., Poelmans, S., Dedene, G.: An architecture for bridging OO and business process modelling. In: 33rd International Conference on Technology of Object-Oriented Languages (TOOLS), Mont-Saint-Michel, France (2000) 132–143
11. Wirtz, G., Weske, M., Giese, H.: The OCoN Approach to Workflow Modeling in Object-Oriented Systems. Information Systems Frontiers **3** (2001) 357–376
12. van der Aalst, W.M.P., de Medeiros, A.K.A., Weijters, A.J.M.M.: Genetic Process Mining. In: Applications and Theory of Petri Nets, 26th International Conference, ICATPN 2005, Miami, USA (2005) 48–69
13. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The ProM Framework: A New Era in Process Mining Tool Support. In: 26th International Conference on Applications and Theory of Petri Nets (ICATPN), Miami, USA (2005) 444–454
14. Reijers, H.A., Limam, S., van der Aalst, W.M.P.: Product-Based Workflow Design. Journal of Management Information Systems **20** (2003) 229–262
15. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal **42** (2003) 428–445